Memory Management - I

Memory is Shared

- In a computing system the CPU is shared between processes (threads).
- It is necessary to keep images of different processes in the main memory[a].
- So the main memory is also shared among different processes (threads).

[a]It will be very costly in terms of time to load the process image during every context switch.

Memory is Shared

- The logical memory space available to each process on a particular system is almost identical.
- It is necessary to map different logical spaces to different parts of the main memory.
- Every logical address generated by the CPU is translated to the corresponding physical memory address.

Memory is Shared

- If the translation of logical address to main memory address is slow, the performance of the system will be poor.
- It is also necessary to protect the image of one process against any attempt to tamper it by another process.
- No user process should be allowed to modify the translation data.

Memory is Shared

- There are also other issues e.g. whether the whole or some parts of a process image should be kept in the main memory at any point of time.
- If the whole image is kept in the main memory, a fewer number of processes can be simultaneously accommodated in the main memory.

Memory is Shared

- If only the currently active parts of the images are kept in the main memory[a], it will be necessary to bring the other parts (data or code) in the main memory as and when required. That requires access to backup store.
- But transferring data from the backup store to main memory is a slower process.

[a] Other parts are in the storage device e.g. disk.

Memory is Shared

- Moreover the main memory may be already full and it will be necessary to free some part of it.
- There should be some protocol to select a block of memory to free.
- A selected block to be removed might have been modified (dirty) and it is necessary to write it back in the backup store.

Memory Fault

- When a process tries to access a logical address that is currently not present in the main memory, there will be a memory exception.
- The process cannot complete the offending instruction and switches to the kernel mode.

Memory Fault

- The kernel initiates the loading of the required portion of the logical space from the backup store to the main memory and the process is suspended.
- Once the required memory is loaded, the process is ready for execution.
- When it is scheduled, it starts running from the offending instruction.

Memory Mapping with Base and Limit Registers

- We start our discussion about logical memory mapping and its protection with the following simple scheme.
- Assume that every process image starts from the same logical address zero (0) and is up to certain high value a_h.
- The whole image is loaded as a single block in the main memory.

Memory Mapping with Base and Limit Registers

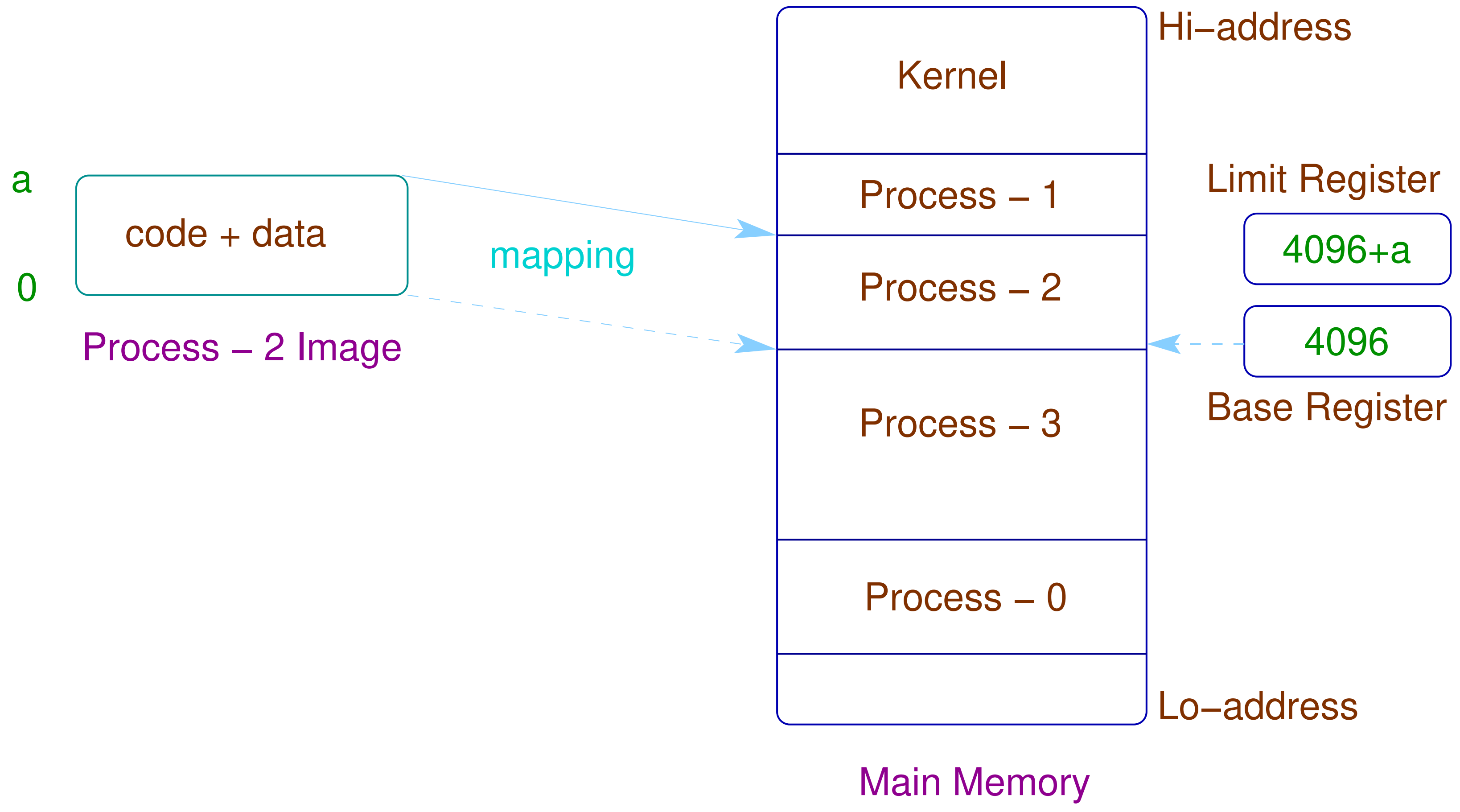
- Two registers are used in this scheme for address translation and protection.
- One register is called the base register that stores the address of the main memory where the process image starts[a].
- The other register is the limit register. It stores the upper limit of the address[b].

[a]Logical address zero (0) is mapped to that.
[b]The logical address a_h is mapped to base + limit.

Memory Mapping with Base and Limit Registers

- Consider a process P_2 whose range of logical address is 0 to a_h.
- It is loaded in the main memory starting from the address 4096.
- The base register is loaded with 4096 and the limit register is loaded with $4096 + a_h$.

Base and Limit Registers



Memory Mapping with Base and Limit Registers

- The CPU while executing code of the process P_2 normally generates address in the range of $0 - a_h$.
- The logical address is added to the base address stored in the base register to get the main memory address.
- The main memory address is compared with the value stored in the limit register.

Memory Mapping with Base and Limit Registers

- If the main memory address exceeds the limit register value, it is a violation of memory that transfers control to the kernel for exception handling.
- User process cannot modify the base and limit registers as that requires privileged instructions. This protects the memory space of one process from another.

Memory Mapping with Base and Limit Registers

- The kernel saves[a] and loads[b] these two registers during a context switching.
- When a process is running in the kernel mode, the base register may be set to zero (0) and and limit register to the limit of the main memory.

[a]For the preempted process.
[b]For the scheduled process.

Memory Mapping with Base and Limit Registers

- The kernel code needs to access the whole main memory for different purpose e.g. loading a process image, using part of memory for IO buffer and copying data from the buffer to the process space etc.
- The architecture may support the kernel to bypass these two registers.

Memory Mapping with Base and Limit Registers

- This simple scheme does not work for a modern computing system due to several reasons.
- The scheme requires contiguous loading of the whole process image.
- It is not clear how shared memory can be implemented using the scheme.

Memory Mapping with Base and Limit Registers

- So communication between processes through shared memory or use of shared library seems to be impossible or very difficult.
- It is also not possible to restrict access in different parts of the process image. For example the code is read-only, but data is not. The code is usually not executable.

Memory Allocation and External Fragmentation

- The kernel keeps a list of main memory blocks occupied by different processes and also a list of available (free) blocks (along with their sizes)[a].
- In our base + limit register scheme, a contiguous block of sufficient size is required to load a process image to start execution.

[a]This can be done through the free blocks itself forming a linked list.

Memory Allocation and External Fragmentation

- After the termination of a process the memory block used by it is released to the kernel pool of free blocks.
- It is possible that there are more than one free blocks of sufficient sizes to accommodate a process image. The natural question is, which one to choose to load the image.

Memory Allocation and External Fragmentation

- There are different suggestions e.g. first-fit, best-fit, worst-fit etc.
- first-fit: chooses the ‘first block’ of proper size from the list of free blocks.
- best-fit: chooses the smallest among the free ‘blocks’ of suitable sizes.
- worst-fit: chooses the largest among the free ‘blocks’ of suitable sizes.

Memory Allocation and External Fragmentation

Each strategy has its justification -

- first-fit - it picks up the first available block of suitable size to load the image, so it is fastest.
- best-fit - it picks up the smallest block to load the image, so the leftover of the free block is of smallest size.

Memory Allocation and External Fragmentation

- worst fit - it pickup the largest block to load the image, so the leftover free block is of largest size.
- People run simulations on sequence of request and release of blocks to measure goodness of different strategies.
- It was observed that first fit and best fit are better strategies compared to worst fit.

Memory Allocation and External Fragmentation

- After some allocation and release of memory blocks, the available blocks may be broken into small pieces.
- Due to this, it is possible that the total volume of free memory is sufficiently large, but there is no free block of suitable size to load a process image.
- This is external fragmentation of memory.

External Fragmentation and Memory Compaction

- One possible solution to external fragmentation is memory compaction i.e. to collect all smaller free blocks to one large free block.
- Memory compaction can be done in run-time by copying process images and modifying corresponding base + limit registers.

External Fragmentation and Swapping Out

- But copying of almost the whole main memory is time consuming.
- Another solution to accommodate a new process in the memory where no free block of suitable size is available is to swap out an existing process image (may be a suspended one) to the backing store e.g. disk.

Swapping of Process Image

- When a scheduler decides a context-switching, it invokes the dispatcher.
- The dispatcher picks the next process from the ready queue.
- If its image is already there in the memory, the execution may start immediately.

Swapping of Process Image

- If it is not, the dispatcher tries to find a free memory block to load the process image.
- If no such free block is available, a process image of suitable size from the main memory is swapped out[a] to the backing store.
- The selected process image is loaded in the newly created free memory block.

[a]Selection of a process for swapping is decided by some algorithm.

Swapping of Process Image

- Swapping out a process and loading one for execution is again time consuming. It is also not easy to select the right process to swap out.
- Even in the swap area of the backing store there will be external fragmentation problem.

Segmented View of Memory

- The logical memory space of a process is divided in several parts e.g. code segment, data segment, stack segment, heap (data segment allocated at run time), shared library, shared memory etc.
- It is possible to divide the logical space into these segments and map each segment to different contiguous main memory locations.

Segmented View of Memory

- Dividing the logical space in smaller segments has its advantages. The required size of memory block for each segment will be smaller.
- This may reduce the possibility of external fragmentation.
- Each segment can have different types of permissions e.g. `r-x`, `rw-`, `r--` etc.

Segmented View of Memory

- There may be shared segments between processes.
- All segments need not be present in the main memory[a].
- In this scheme a logical address has two components, a segment address(l_s) and an offset (l_o) within a segment.

[a]Loaded on demand.

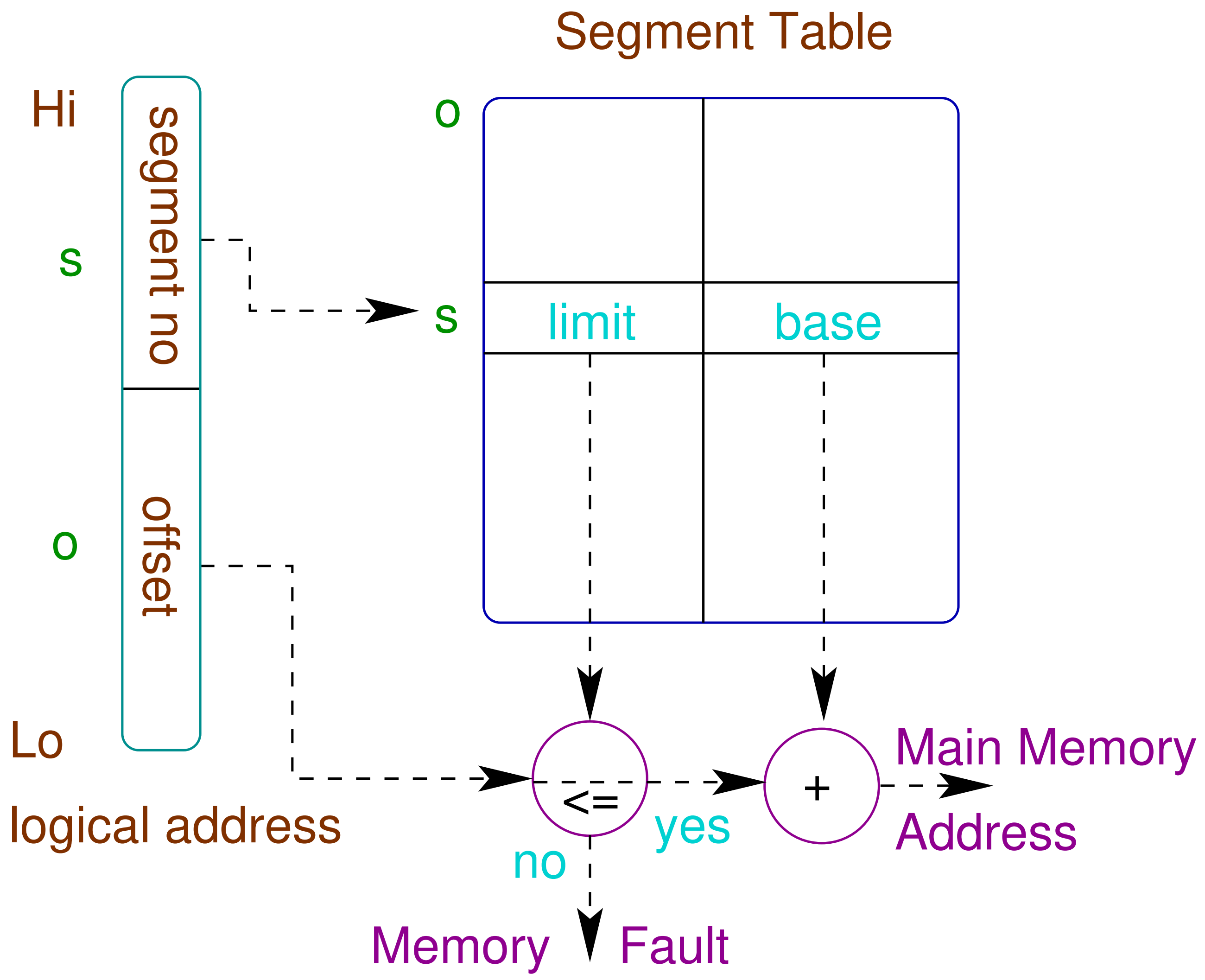
Segmented View of Memory

- But then each segment requires its own base and limit registers.
- There is a segment table per process. The segment address part of the logical address (l_s) is used to access the segment table.
- Each entry in the segment table has a base address of the segment in the main memory (s_b) and its size or limit (s_l).

Segmented View of Memory

- If the segment offset (l_o) of the logical address is less than or equal to the segment limit ($l_o \leq s_l$), then the segment base address is added to the offset ($s_b + l_o$) to get the main memory address.
- Otherwise there will be memory access fault.

Segment Table Translation



Segmented View of Memory

- An obvious question is where to store the segment table.
- If the size of the table is small, it can be stored in the memory management unit (MMU) of the hardware.
- But a large segment table should be memory resident.

Segmented View of Memory

- If the segment table is stored in the memory, then for each logical memory access there are two main memory access.
- The first one is to access the segment table in the memory, and the second one is to access the actual data or instruction.
- This will slowdown the process considerably.

Segmented View of Memory

- One way to achieve faster address translation is to keep relevant segment table entries in a fully associative cache memory[a] in the memory management unit.
- Most of the time the translation will be through the entries present in the cache and will be fast.

[a]Not to be confused with the cache memory in the memory hierarchy.

Segmented View of Memory

- If a segment entry is not available in the segment table cache, the segment table in the main memory is accessed for translation.
- This entry will also be loaded in the segment table cache to make subsequent translations faster[a].

[a]We shall discuss about the cache in detail in connection to paging.

Segmentation to Paging

- Though segmentation divides the logical address space, it cannot avoid external fragmentation.
- Also segments are of different sizes which makes loading a segment, and housekeeping of free and occupied memory slots more difficult.

Segmentation to Paging

- An alternate scheme called paging has become more popular. All pages are of equal size.
- There is no external fragmentation in paging, but there may be loss of memory due to partly occupied page, known as internal fragmentation.

Basic Paging

- The main memory is divided into fixed size blocks known as page frames.
- The logical address space is divided in equal size blocks called pages.
- When a process image is loaded, its pages are loaded in free page frames of the main memory.

Basic Paging

- For each process there is a kernel data structure called a page-map table or page table[a].
- The page table of a process translates a valid logical address to the main memory address.

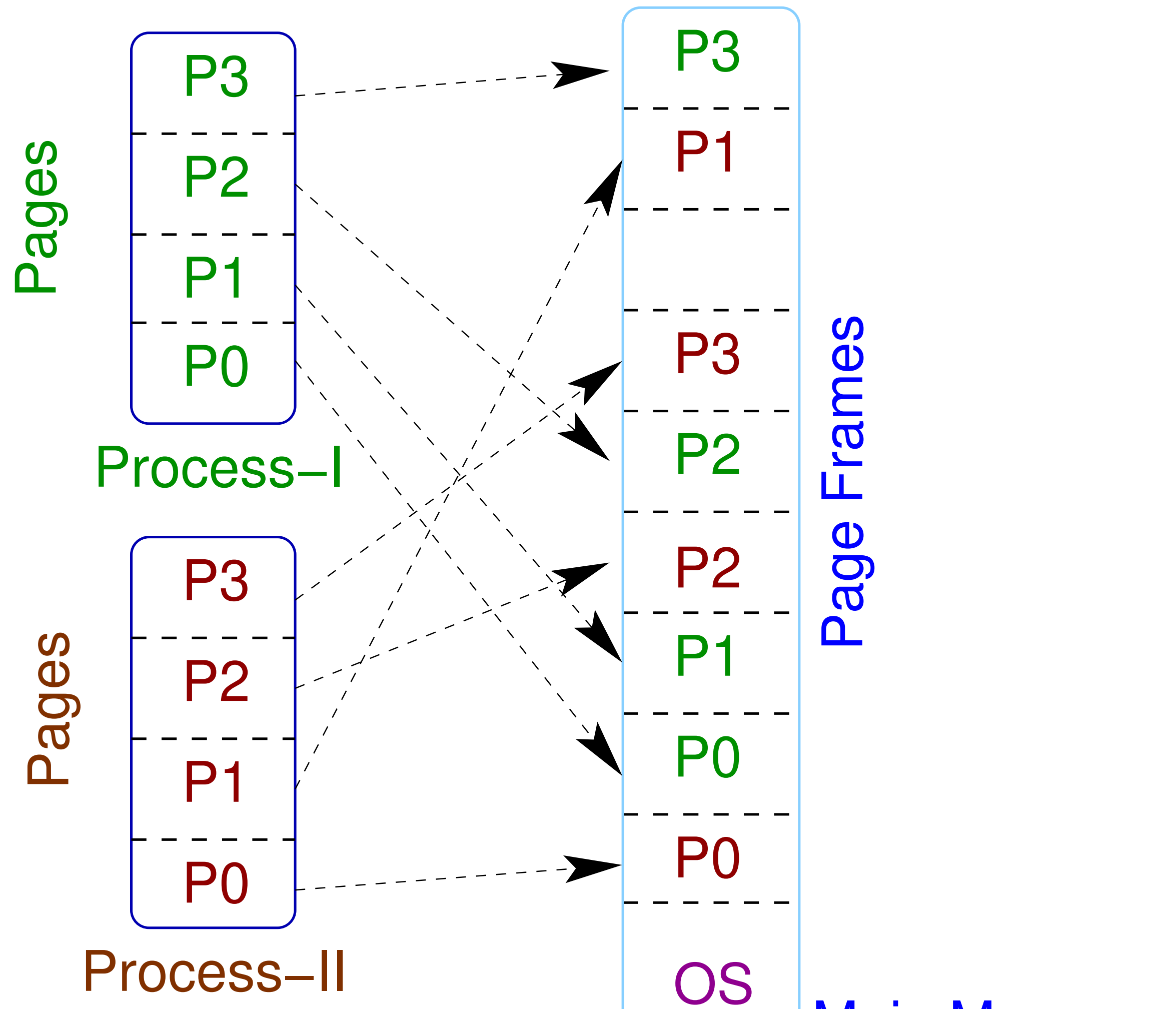
[a]There are schemes where a global page table for the entire system is used.

Page Fault

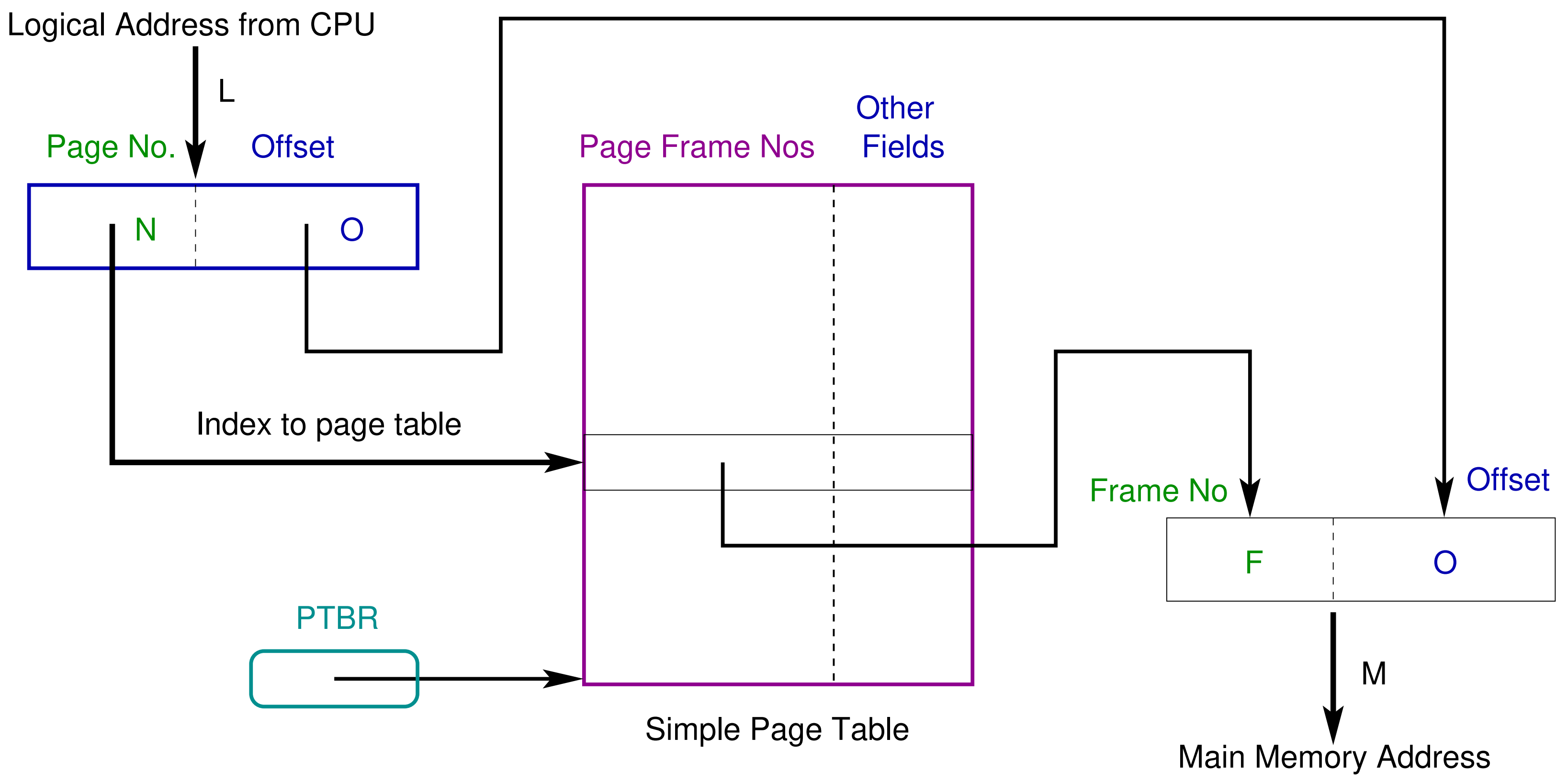
- If the logical address is invalid or the page is not present in the main memory, then it can be detected from the page table.
- This event generates a page fault exception. Different actions are taken by the OS kernel depending on the situation[a].

[a]If the page is invalid it is a memory violation. Otherwise an empty page frame may be allocated to the process. The new page frame may be loaded from the backing store e.g. file system or swap area etc.

Page Mapping



Page Mapping



Page Table

- In the simplest form of page table mapping a logical address has two parts - a page number and an offset within the page.

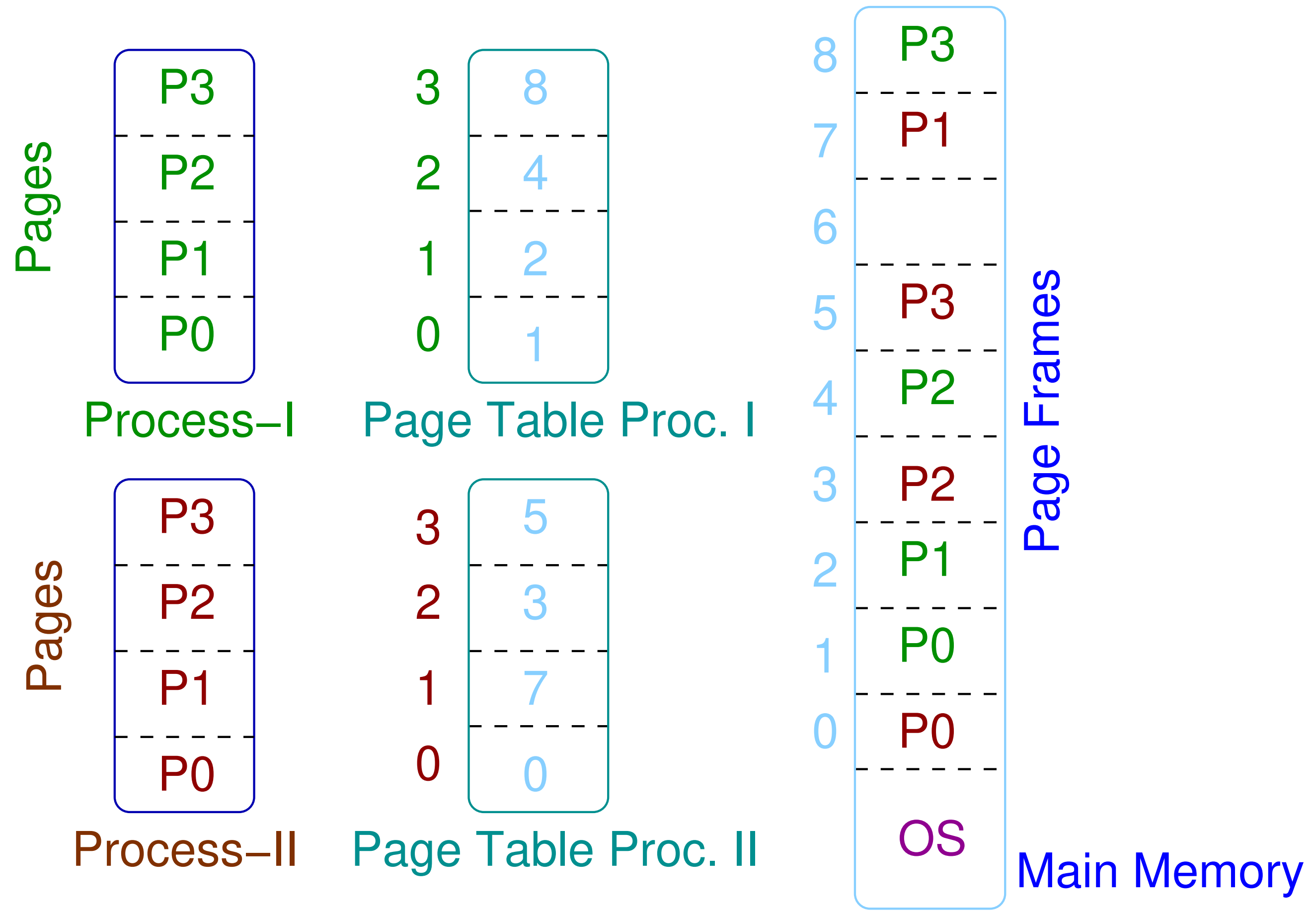
Logical address:

Page Number	Offset

- The page number is an index to the page table that stores the base address of the page frame holding the page[a].

[a]Provided it is valid and present.

Page Mapping



Page Mapping: an Example

- Let the logical address be 32-bit $(a_{31} \cdots a_0)$, and the page-frame size be 4-KB.
- The bits $a_{11} \cdots a_0$ specifies the offset within a page.
- The page number is specified by $a_{31} \cdots a_{12}$ i.e. there are $2^{20} = 1\text{M}$ pages.

Page Mapping: an Example

- Let the size of each entry of the page table be 32-bits, of which 20-bits specify the page frame number.
- The total size of the page table for every process is 4 MB.
- This is too large a space for meta data per process. We shall see solutions for this.

Page Mapping: an Example

- Consider a logical address 0x12345678.
- The offset within a page is 0x678 = 0110 0111 1000B.
- The page number is 0x12345 = 0001 0010 0011 0100 0101.

Page Mapping: an Example

- Let the page table entry corresponding to this page number be 0x87654321. The least significant 20-bits specifies the base address[a] of the page frame.
- Let the page frame base address be 0x54321000.
- The main memory address corresponding to the logical address is 0x54321678.

[a]Most significant 20-bits.

Page Mapping: an Example

- Other 12-bit of the page table entry are used to store different information about the page e.g. page valid, page present, page CoW, `r-w-x` permissions etc.
- If the page is not present in main memory, the lower order 20-bits may store information about its location in the backup store.

Page Table: Linux on x86-64

- The logical address of x86-64 architecture is 48-bit $(a_{47} \cdots a_0)$.
- Common page size is 4 KB $(a_{11} \cdots a_0)$[a].
- The page number is specified by $a_{47} \cdots a_{12}$.
- Every page has a 64-bit entry in the page table.

[a]Other page sizes are also possible.

Page Table: Linux on x86-64

- The number of pages per process is $2^{36} = 64G$!
- So the size of page table per process is 512 GB.

Page Table: Linux on x86-64

- Given a logical address (A) (say of a variable), we can find its page number by $p_n = A/p_s$, where p_s is the page size.
- The offset of the corresponding page table entry is $t_o = p_n \times e_s$, where e_s is the entry size.

Advantages of Paging

- Allocation of a page in page frames is simple as each page is of same size.
- There is no external fragmentation. But a few pages per process may not be completely full (internal fragmentation)[a].

[a] Any access to that logically empty part is difficult to restrict.

Advantages of Paging

- Shared memory among processes is easy to implement in a paged memory management system. Shared main memory frames can be attached to logical pages of different processes.
- If the code is not self-modifying (re-entrant) it can also be shared. OS kernel can ensure that the permission for pages of shared code is read and execute.

Advantages of Paging

- In general access restrictions can be enforced for every page through the page table.
- A page table entry not only contains the mapping to the main memory address, but also bits are kept for access permissions and other information.

Address Translation by Page Table

There are several issues in connection to the page table and address translation.

- The first question is where to keep the page table and how to organize it.
- Then what will be the time penalty for address translation and how to reduce it.

Page Table in Hardware

- If the logical address space is small and there are smaller number of page frames, the page table can be stored in the memory management unit (MMU).
- Consider the the case of PDP-11 (DEC) where the logical address was 16-bit, and the page size was 8 KB.
- So there are only eight (8) page frames.

Page Table in Hardware

- The 8-entry page table was stored in the hardware, which will be saved and loaded by the Kernel during contest- switching[a].
- In this case the address translation will be fast, and the penalty is little.

[a]Must be done by privileged instructions.

Large Page Table per Process

- But in a modern CPU the size of a page table is much larger. Assuming the size of page frame to be 4 KB.
- On a 32-bit architecture, the number of pages are 2^{20}.
- On a 48-bit logical address e.g. x86-64[a], the number of pages are 2^{36}.

[a]Higher order 16-bits are not used.

Large Page Table per Process

- In case of 32-bit architecture, assuming 4-bytes for each page table entry, the size of the page table (per process) is 4 MB.
- In case of 48-bit logical address, assuming 8-byte for each page table entry, the size of the page table is $\frac{1}{2}$ TB.

Page Table in Memory

- A page table of such a large size cannot be stored in the MMU hardware.
- Even if the whole page table is stored in the memory, it will occupy a large portion of it.
- For 32-bit address space the full table occupy 1K page frames, and for 48-bit address space it will occupy 128 M page frames, per process.

Page Table in Memory

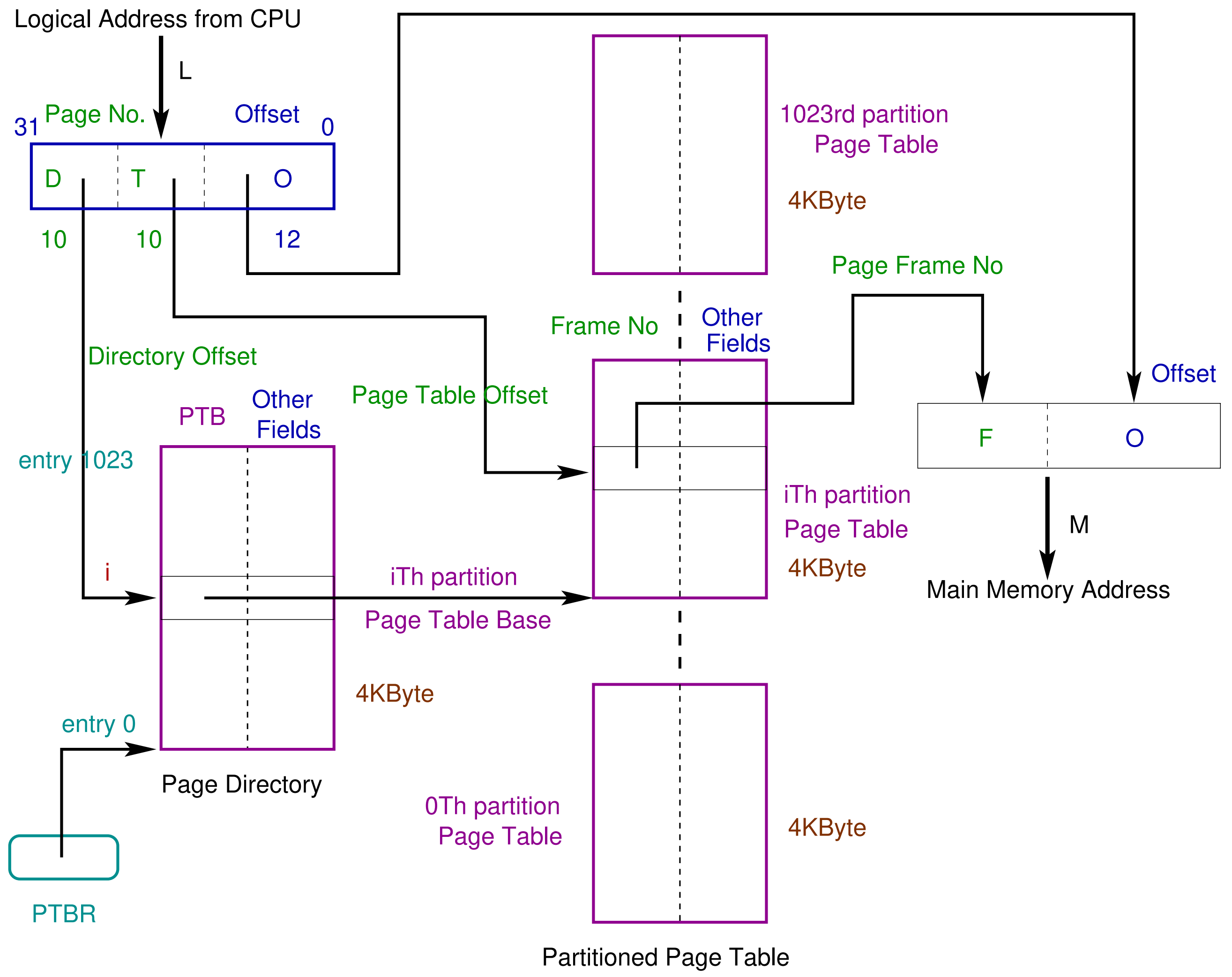
- Large page tables are divided into pages.
- Every page of the page table for a process need not be valid[a].
- Every valid page of the page table need not be present in the main memory for the whole running time.

[a] There are large 'holes' in the logical address space.

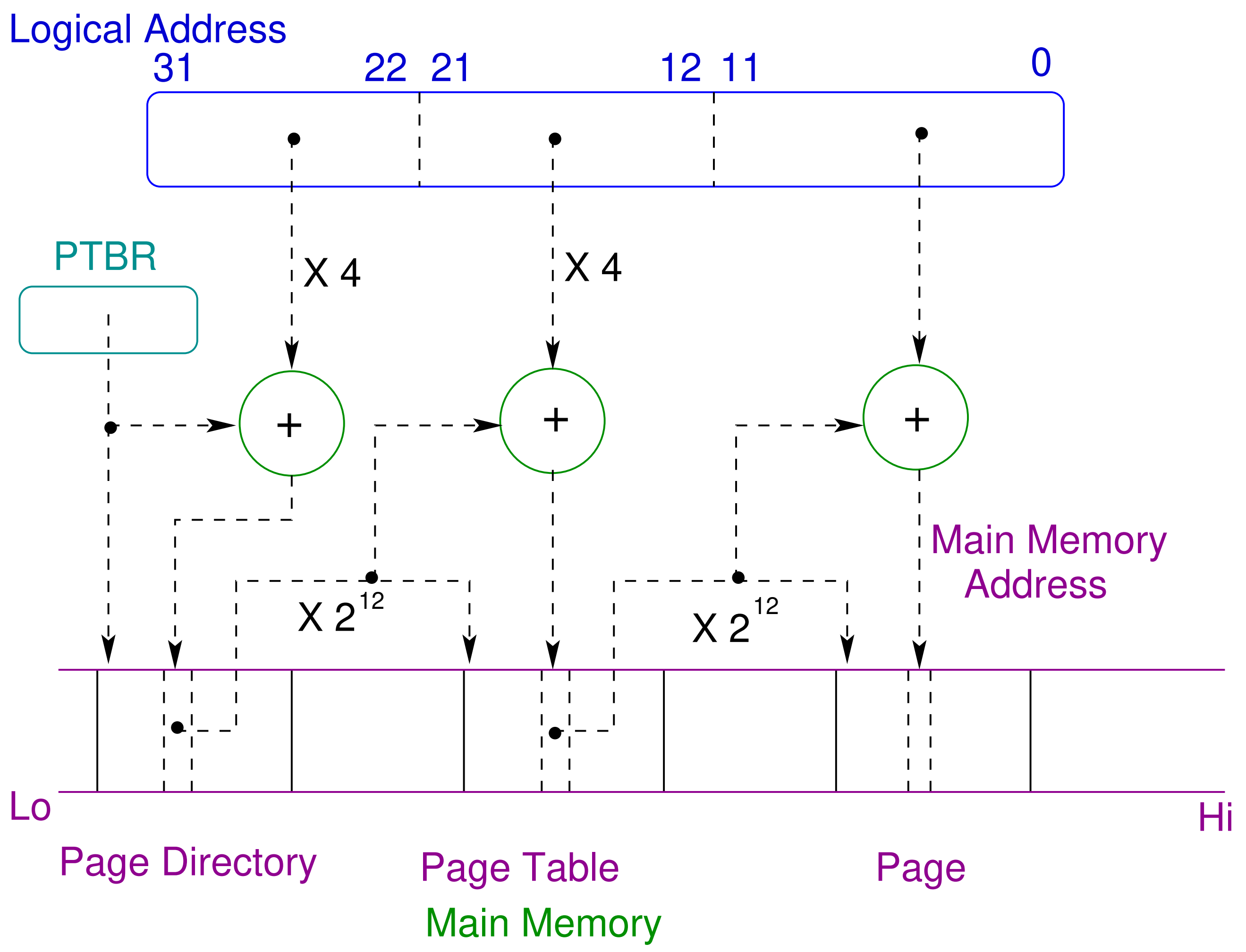
32-bit Two-Level Paging: an Example

- The 32-bit logical address is translated to main memory address using a 2-level page table.
- The top level is called a page directory. Its size is 4KB and there are 1K entries each of size 4B.

32-bit Address Translation



32-bit Address Translation



32-bit Two-Level Paging: an Example

- The base address of the page directory is stored in a MMU register called the page directory base register (PDBR)[a].
- The higher order 10-bits of the logical address provides the offset within the page directory ($PDBR + 4 \times a_{31} \cdots a_{22}$).
- Each page directory entry corresponds to 4 MB of logical space.

[a]It is called `cr3` in IA-32 architecture.

32-bit Two-Level Paging: an Example

- 20-bits of a 32-bit bit entry of the page directory provides the base address $(d_{19} \cdots d_0 \overbrace{0 \cdots 0}^{12})$ of a partition of the page table, where each entry is of size 4B.
- The offset within this partition of the page table comes from the next 10-bits of the logical address $(2^{12} \times d_{19} \cdots d_0 + 4 \times a_{21} \cdots a_{12})$.

32-bit Two-Level Paging: an Example

- 20-bits of a 32-bit bit entry of the page table provides the base address $(t_{19} \cdots t_0 \overbrace{0 \cdots 0}^{12})$ of the required page.
- The offset within the page comes from the last 12-bits of the logical address $(2^{12} \times t_{19} \cdots t_0 + a_{11} \cdots a_0)$.

32-bit Two-Level Paging: an Example

- Both from the page directory and from the page table 20-bits of each entry are used as the base addresses[a] of a page frame in the main memory, provided the portion of the page table and the page are present.
- Otherwise, if the page is valid, it may contains information about the location of the page in the backup store.

[a]Aligned to 4KB boundary.

32-bit Two-Level Paging: an Example

- Remaining 12-bits of each entry stores other information e.g. the page is valid, page or page table is present, CoW and other permissions etc.
- A page fault exception is generated if either the page is not valid or not present.
- Following the exception, necessary actions are taken by the Kernel.

Note

- The address of an instruction or of a data may generate a page fault exception.
- If it is a real memory access violation, the process may be terminated.
- But there are other possibilities where a new page frame may be allocated to the process. And it may be loaded with data from the backup store.

Note

- At times it may be necessary to suspend the process during the data read.
- But then the process is to be restarted from the instruction that has generated the page fault.
- That requires some housekeeping by the hardware and the page fault handler.

Note

- It is also possible that no free page frame is available to load the new page.
- It will be necessary to remove some of the existing pages to create space.
- But then the question is which page to remove, and what is to be done if the page is dirty.
- There are page replacement policies.

Advantages of Hierarchy

- The hierarchical page table organization has tremendous advantage in terms of space usage per process.
- Only the top-level of the page table[a] must be present in the main memory.
- Other levels may be absent (invalid) altogether or may be at the backup store and loaded on demand.

[a]Page directory for 2-level or PML4E for 4-level for each process.

Problem of Hierarchy

- Each logical memory access requires three (3) main memory access in case of 2-level page table and five (5) main memory access in case of 4-level page table.
- This is simply unacceptable.
- The MMU architecture provides hardware support to solve the problem.

Spatial and Temporal Locality of Access

- It is well known that access to memory locations often changes slowly over time.
- Most often the current-instruction is not a jump or a branch. So instruction execution takes place in a spacial locality.
- Most of the time spent in execution of a code are in loops. The same code is executed again and again - it is temporal locality.

Spatial and Temporal Locality of Access

- Data access also has a pattern and it is claimed that this too satisfies the principles of locality.
- The number of pages accessed by a process during its CPU time slice is not too large.
- Often a small portion of the page table is used for address translation during this period.

Translation Look-aside Buffer (TLB)

- The MMU architecture provides a fully-associative or set-associative cache known as a translation look-aside buffer (TLB). It stores the translation of most relevant logical pages.
- Most often the logical addresses generated by the CPU is translated to the main memory addresses using the TLB.

Translation Look-aside Buffer (TLB)

- If the entry corresponding to the logical page is not present in the TLB (miss), the page table is accessed for translation[a].
- The translation entry (page number, frame base address, protection etc.) from the page table is also loaded in some free slot of the TLB.

[a]This may be done by the hardware or by the OS kernel in response to an exception may be called a soft page fault.

Translation Look-aside Buffer (TLB)

- If no free slot of the TLB is available, some existing entry is removed to accommodate the new translation.
- The replacement policy may be least recently used (LRU), random, or round-robin (RR).

Context-Switching

- At the time of context-switching the PTBR corresponding to the page table of the preempted process is saved.
- The new value of the page directory base address corresponding to the scheduled process is loaded in the PTBR.
- TLB entries are invalidated as they correspond to the preempted process.

Context-Switching

- Some TLB stores an identifier of a process[a] along with its translation entries.
- In such a case the logical address must contain the identifier of the generating process.
- A TLB entry is valid only if the current process identifier matches with the identifier of the TLB entry.

[a]This need not be the PID given by the OS kernel.

Context-Switching

- In a TLB with process identifier, it is not necessary to invalidate entries during context-switching.

Inverted Page Table

- The logical address space of a modern processor is very large. So the possible number of pages are also large.
- Each page has an entry in the page table (for each process).
- Many of these entries are actually invalid as there is no real page corresponding to it.

Inverted Page Table

- But large number of entries makes the size of page table per process rather large.
- Hierarchical page table organization solves this problem to some extent where lower label tables may be absent.
- An Inverted page table is an alternate solution.

Inverted Page Table

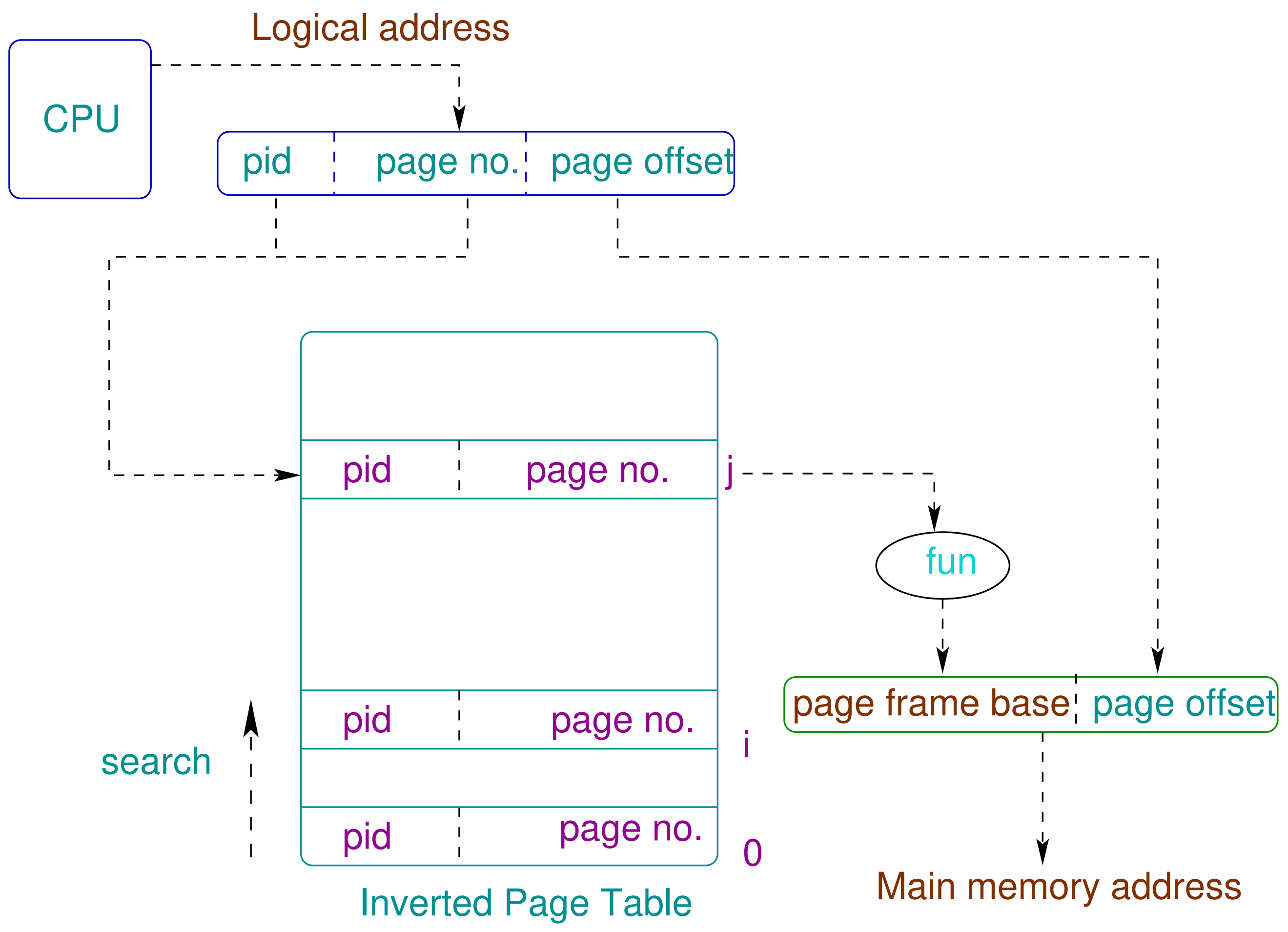
- It is a global page table with each entry corresponds to a page frame of the main memory.
- Each valid entry of the table contains a logical page number and the corresponding process identifier.
- The base address of the page frame can be derived from the index of the page table[a]

[a]base address = index$\times k + c$.

Inverted Page Table

- The table is searched for a logical address along with the process identifier. The search naturally takes a long time.
- But most of the time the address translation takes place through the TLB cache and the table search is not necessary.
- It is difficult to implement shared memory on an inverted page table.

Inverted Page-Table



Memory Mapped File

- A disk file is accessed using IO system calls like `open()`, `read()`, `write()` etc.
- But a file or a portion of it can also be mapped to the main memory and attached to the logical address space of one or more processes.

Memory Mapped File

- Thereby a file IO can be done by access to the memory regions where it is attached.
- Data can be read, written and even execute. File can also be used for inter process communication.
- Following are a few examples.

Memory Mapped File: Read

```
/*
 * memoryMapFile1.c++ maps a file to the address
 *                    process in read only mode
 * $ ./a.out ./inData
 */
#include <iostream>
using namespace std;
#include <stdlib.h>
#include <sys/types.h>
```

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

int main(int ac, char *av[]){
    int fd, size;
    char *mapP;

    if(ac < 2){
      cerr << "File path not specified\n" ;
      exit(1);
    }
    fd = open(av[1], O_RDONLY);
    if(fd == -1){
      cerr << "File open error\n";
      exit(1);
    }
    size = sysconf(_SC_PAGE_SIZE);
    mapP = (char *)mmap(0, size,  PROT_READ, MAP_
    if(mapP == MAP_FAILED){
      cerr << "Map failed\n";
```

```
        exit(1);
    }
    cout << mapP << endl;
    close(fd);

    return 0;
}
```

Memory Mapped File: Read

- The file name supplied as the command line argument contains a string of characters.
- It is opened as usual in read mode, attached to the logical address space by the `mmap()` with with the read permission.
- The specified length of the mapping is the page size of the memory.

Memory Mapped File: Read

- It is a private mapping[a].
- The offset is zero (0) i.e. the mapping starts from the beginning of the file.
- The mapping address of the logical address space is unspecified (`NULL`) and is decided by the kernel.

[a]Modifications, not applicable for this example, is not reflected in the original file.

Memory Mapped File: Read

- The call to `mmap()` returns the starting address of the attached memory location.
- The file can be read using this address.
- Finally we unmap and close the file.

Demand Paging

- The Kernel may load the complete process image while creating it. But that may lead to poor utilization of main memory for a large process.
- An alternative is to make the lazy loading of the process pages on demand and also in anticipation.

Demand Paging

- But that requires swapping pages back and forth between the main memory and the backup store.
- There are mainly two places of the backup store from where this swapping takes place. They are the file system and the swap area.
- Demand paging requires some support from hardware architecture.

Demand Paging: Page Table

- A valid page of a process may not be present in the main memory.
- Each page table entry may have another bit (present) to indicate the presence or absence of a valid page.
- It is also necessary to save the information about the location of the page in the backup store.

Demand Paging: Page Table

- If a page is absent from the main memory, the bits used to store the page-frame number in the entry are available.
- These bits may be used to store the block number of the backup store where the page was swapped out or to be loaded from.

Demand Paging: Page Fault

- Page faults of a demand paging system are of at least of two different types.
- The page fault may be due to an attempt to access an invalid page by the process[a].
- The page is valid but not present in the main memory.

[a]It may be invalid due to several reasons - (i) not a valid address, (ii) attempt to write on a read-only page etc.

Demand Paging: Page Fault

- On any page-fault the state of the CPU[a] is saved along with the address of the offending instruction and the offending address of data. The control is then transferred to the kernel.
- If the offending address is invalid, the process is terminated.

[a]The state before the execution of the current instruction.

Demand Paging: Page Fault

- If the address is valid but the corresponding page is not present in the main memory, following actions are initiated by the page-fault handler.
- A page-frame is obtained from the the list of free page-frames[a].

[a] If there is no page-frame free, some occupied frame is to be replaced.

Demand Paging: Page Fault

- The operation of loading the page, may be spread over several disc blocks, from the file system or the swap area is initiated.
- As the disk IO is a much slower operation, the process is suspended.
- Once the page is loaded in the frame, the kernel updates the page-table data structure and changes the process state to ready.

Demand Paging: no Free Page-frame

- If there is no free page-frame available, one of the occupied frames is to be freed. The question is which one.
- Different page replacement algorithms are proposed. But there are two essential issues.

Demand Paging: no Free Page-frame

- The kernel must decide the number of pages for a process to be present in the main memory. This determines the degree of multi programming.
- The kernel also have a policy to select a page frame to replace when there is no free frame available. The policy may be local to a process or may be system wide.

Demand Paging: Replacing a Dirty Page

- The frame selected for replacement may be dirty (modified)[a] i.e. its last image in the disk contains old data.
- So it is necessary to write it back in the disk. It amounts to writing a page and reading the demand page[b].

[a] Indicated by the dirty bit of the page table entry.

[b] The process can be expedited using a free page frame list.

Page Replacement and Performance

- Page replacement has an impact on the performance on the computing system.
- If the replaced page is referenced immediately, there will be a page fault and the replaced page is to be brought back to memory by replacing another page.
- Very frequent page replacement may lead to a phenomena called thrashing.

Page Replacement and Performance

- Let the memory access time be m and page-fault handling time be f.
- If the probability of page-fault is p, $0 \leq p \leq 1$, the average memory access time $a = (1-p)m + f$
- Typically the value of m is in the order of 10's of nanoseconds and f is of the order of milliseconds.

Page Replacement and Performance

- If we take $m = 20$ ns and $f = 10$ ms, then the average access time is $a = 20(1 - p) + 10^7 p \approx 20 + 10^7 p$.
- If we wish to keep the degradation of memory access time for demand paging within 10% of the actual memory access time, then $10^7 p = 2 \Rightarrow p = 2 \times 10^{-7}$.

Page Replacement Algorithm: FIFO

- The oldest page brought into the memory will be replaced.
- A queue of page frames in the order of their loading is maintained.
- When required, a page from the front of the queue will be replaced.

Page Replacement Algorithm: FIFO

- The problem is, an active page loaded long back may be replaced. An immediate subsequent access to it will give rise to a new page fault.
- The algorithm shows a peculiar behavior known as Belady's anomaly. The number of page faults increases for some sequence of page references even with the increase of page frame numbers.

Page Replacement Algorithm: FIFO Example

Page references: 7 2 3 1 2 5 3 4 6 7 7 1 0 5 4 6 2 0 3 0 1,
No of Page frames: 3

Frames			Ref. Pages	Fault
7			7	1
7	2		2	2
7	2	3	3	3
1	2	3	1	4
1	2	3	2	4
1	5	3	5	5
1	5	3	3	5
1	5	4	4	6
...	...	...	...	...

Page Replacement Algorithm: Optimal

- If the complete sequence of page references is known, then for a fixed number of page frames, there is an algorithm that gives lowest number of page faults.
- A page that will not be referenced for the longest time in future is replaced.

Page Replacement Algorithm: Optimal Example

Page references: 7 2 3 1 2 5 3 4 6 7 7 1 0 5 4 6 2 0 3 0 1,
No of Page frames: 3

Frames			Ref. Pages	Fault
7			7	1
7	2		2	2
7	2	3	3	3
1	2	3	1	4
1	2	3	2	4
1	5	3	5	5
1	5	3	5	5
1	5	4	4	6
...	...	...	...	...

Page Replacement Algorithm: Optimal

- The optimal page replacement algorithm is not useable simple because the future frame references depend on data.
- In case of global replacement policy it also depend on page references of other processes on the system.
- It is mainly used to compare other replacement algorithms.

Page Replacement Algorithm: LRU

- The LRU (least recently used) algorithm works on assumption that a page referenced long back may not be referenced in near future.
- Each page frame may be associated with a time stamp. When a page is referenced, the time stamp is updated.

Page Replacement Algorithm: LRU

- The page with the oldest time stamp is replaced when required.
- The main problem of implementation is the fast update of time stamp on each reference to a page, and search for the page with the oldest time stamp.
- Its implementation requires architectural support.

Page Replacement Algorithm: LRU

Page references: 7 2 3 1 2 5 3 4 6 7 7 1 0 5 4 6 2 0 3 0 1,
Page frames: 3

Frames			Ref. Pages	Fault
7_0			7	1
7_1	2_0		2	2
7_2	2_1	3_0	3	3
1_0	2_2	3_1	1	4
1_1	2_0	3_2	2	4
1_2	2_1	5_0	5	5
3_0	2_2	5_1	3	6
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

LRU Implementation: I

- A time stamp field is associated with each page table entry. It is updated every time the page is referenced.
- There may be a logical clock or counter available in the CPU. Which is incremented on every memory reference and its value is written in the time stamp field of the corresponding page.

Note

- Most address translations are through the TLB.
- What happens when there is a context switch.
- It takes time to find the LRU page when a page replacement is required[a].

[a] The page with oldest time stamp may not be in the main memory.

LRU Implementation: II

- A stack-like data structure may be used to keep track of page usage.
- The bottom of the stack holds the LRU page, and the top of the stack holds the most recently used page.
- When a page is referenced, it is taken out from the middle of the stack and put on the top of it.

LRU Implementation: II

- It is costly to update the page reference stack and makes it useless without any architectural support.
- But no search is required to locate the LRU page during a page replacement.
- MMU Architecture provides a very limited support in the form of reference bit(s), insufficient for LRU implementation.

Reference Bit(s)

- There is a reference bit with every entry of the page table. They are set to zero (0) when a process is created.
- Every time a page is referenced, the corresponding reference bit is set to one (1).
- The simplest page replacement algorithm will replace a page of reference bit zero (0).

Better Ordering Using Reference Bits

- The kernel may maintain a list of 8-bit counters (C_k) for each valid page P_k of a process. Counters are initialized to 0000 0000 when a process is created.
- These data structure is updated after a regular interval (I_j) of the order of the CPU time quantum of a process[a].

[a]During context switching.

Better Ordering Using Reference Bits

- At the end of the interval I_j, the counter C_k is shifted right by 1-bit[a] and the reference bit of the page P_k is copied to the most significant bit (b_7) of C_k.
- The counter C_k maintains the reference history of P_k through the last eight intervals with a higher weight for more recent intervals.

[a] The least significant bit is discarded.

Better Ordering Using Reference Bits

- If there are n pages, $\{P_0, \cdots, P_{n-1}\}$, then a page corresponding to $\min\{C_0, \cdots, C_{n-1}\}$ is a 'LRU' page.
- There may be several pages with same C_k value and which one to replace is to be decided.
- It also takes time to search for the least C_k from the page reference list.

Modified FIFO using Reference Bit

- The reference bit can be used to modify FIFO algorithm in a simple way.
- If a page at the head of the FIFO queue (circular), but its reference bit is one (1), it is be given a second chance.
- The reference bit is cleared and the page is brought to the rear of the queue by advancing the head pointer.

Modified FIFO using Reference Bit

- A page is replaced if it is at the head of the queue and its reference bit zero (0).
- In the worst case when the reference bits of all pages in the queue are ones (1), the head pointer traverse the whole list to locate a replaceable page, the first one at the beginning.

FIFO using Reference and Dirty Bit

- This algorithm can be modified further using the reference and dirty bits, (r, d).
- There are four (4) possible situations and the best page to replace is the one where $(r = 0, d = 0)$.

Free-Page Pool

- The kernel may maintain a steady free page frame pool.
- The page on demand will be loaded in one of the free pages frames from the pool.
- The selected replaceable page will go back to the free page pool after write-back if necessary.

Bibliography

1. Operating System Concepts by Abraham Silberschatz, Peter B Galvin & Gerg Gagne, 9th ed., Wiley Pub., 2014, ISBN 978-81-265-5427-0.
2. Operating Systems: Three Easy Pieces by Remzi H. Arpaci-Dusseau & Andre C. Arpaci-Dusseau Pub. Arpaci-Dusseau Books, LLC, 2008-19.
3. Beginning Linux Programming by Neil Mathew & Richard Stones, 3rd ed., Wiley Pub., 2004, ISBN 81-265-0484-6.
4. Understanding the Linux Kernel by Daniel P Bovet & Marco Cesati, 3rd ed., O'Reilly, ISBN 81-8404-083-0.
5. `https://github.com/0xAX/linux-insides/blob/master/Theory/Paging.md`